VarGAN: Adversarial Learning of Variable Semantic Representations

Yalan Lin, Chengcheng Wan, Shuwen Bai, Xiaodong Gu

Abstract—Variable names are of critical importance in code representation learning. However, due to diverse naming conventions, variables often receive arbitrary names, leading to long-tail, out-of-vocabulary (OOV), and other well-known problems. While the Byte-Pair Encoding (BPE) tokenizer has addressed the surface-level recognition of low-frequency tokens, it has not noticed the inadequate training of low-frequency identifiers by code representation models, resulting in an imbalanced distribution of rare and common identifiers. Consequently, code representation models struggle to effectively capture the semantics of low-frequency variable names. In this paper, we propose VarGAN, a novel method for variable name representations. VarGAN strengthens the training of low-frequency variables through adversarial training. Specifically, we regard the code representation model as a generator responsible for producing vectors from source code. Additionally, we employ a discriminator that detects whether the code input to the generator contains low-frequency variables. This adversarial setup regularizes the distribution of rare variables, making them overlap with their corresponding high-frequency counterparts in the vector space. Experimental results demonstrate that VarGAN empowers CodeBERT to generate code vectors that exhibit more uniform distribution for both low- and high-frequency identifiers. There is an improvement of 8% in similarity and relatedness scores compared to VarCLR in the IdBench benchmark. VarGAN is also validated in downstream tasks, where it exhibits enhanced capabilities in capturing token- and code-level semantics.

Index Terms—pre-trained language models, variable name representation, identifier representation, generative adversarial networks

1 Introduction

Code representation models, such as CodeBERT [6], have gained widespread adoption as representations of source code due to their ability to capture complex semantic and syntactic features of code. Through pre-training on large-scale code corpora with unsupervised objectives, they have made significant improvements in the state-of-the-art code intelligent tasks and have facilitated the development of new tools and applications for software analysis.

One of the most crucial parts of code representation is the ability to capture accurate semantic representations of identifiers [25], [42]. Identifiers play a critical role in code semantics. They constitute the main proportion of code and serve as important semantic cues for code comprehension [34], [33]. Meaningful and descriptive identifiers can greatly enhance the readability of code [33], thereby facilitating semantic learning by code representation models.

Nonetheless, the representation of source code identifiers presents noteworthy challenges. Unlike natural languages, source code variables are named in an arbitrary manner [14]. Programmers adhere to diverse programming conventions and personal preferences, leading to various formats and styles of variable names. Consequently, a considerable number of low-frequency identifiers emerge, referring to identifiers with infrequent occurrences. This phenomenon gives rise to notable long-tail and out-of-vocabulary (OOV) issues within the context of source code [46], [14].

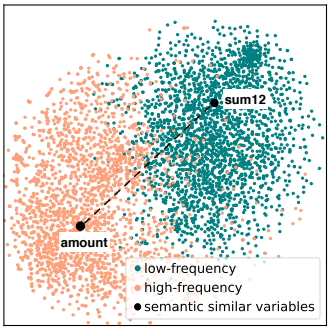


Fig. 1: T-SNE visualization of variable name representations learned by CodeBERT

There have been numerous attempts to resolve this problem such as the utilization of unknown tokens (UNK) and Byte Pair Encoding (BPE) [31]. While these approaches have successfully tackled the surface-level representation of low-frequency tokens, they have not noticed another significant problem: imbalanced training of low- and high-frequency identifiers by PLMs. Low-frequency identifiers occur rarely in the code corpora, thus they have fewer opportunities for training and movement compared to those high-frequency identifiers. Take the variable name `sum12` as an example. Though both its subtokens `sum` and `12` appear frequently in the corpora, their combination has a relatively rare occurrence, making it challenging for the code representation model to accurately capture its semantics. Consequently, the learned vector space exhibits bias towards more common identifiers, leading to an imbalanced distribution of vectors

- *Y. Lin and X. Gu are with the School of Software, Shanghai Jiao Tong University, Shanghai, China. E-mail: {linyalan, xiaodong.gu}@sjtu.edu.cn*
- *C. Wan is with the Software Engineering Institute, East China Normal University, Shanghai, China. E-mail: ccwan@sei.ecnu.edu.cn*
- *S. Bai is with the Department of Computer Science, East China University of Science and Technology, Shanghai China, China. E-mail: shuwenbai2023@163.com*
- *X. Gu is the corresponding author.*

for low- and high-frequency identifiers. The distances in the vector space cannot accurately reflect the correlation between variables. Figure 1 visualizes the representations of Java identifiers obtained by CodeBERT. The data points in orange correspond to high-frequency variables, whereas the points in green represent low-frequency variables. It can be seen that there is an evident discrepancy in the distribution between low- and high-frequency variable names. The overall distribution is cone-shaped, with low-frequency identifiers more concentrated at the head of the cone, while high-frequency identifiers are dispersed along its tail. Even if two low- and high-frequency identifiers are semantically similar, they may exhibit significant distance in the vector space, such as `sum12` and `amount` in Figure 1. A more profound study of this phenomenon can be found in our experiments of RQ1 (Section 4.1).

On the other hand, low-frequency variable names play a crucial role in code comprehension, given that code representation models heavily depend on the semantic cues encoded in identifiers [27]. Removal or insufficient training of low-frequency variables could impact the mo coarse-grained requirements and lack relevant resources.del's performance significantly [43]. For instance, if `sum12` is used as a function name and is removed directly, comprehending this function could be significantly challenging due to the absence of essential function name information. Therefore, it is essential to enhance the model's training process to better comprehend and handle low-frequency variable names.

To tackle this problem, we propose VarGAN, an adversarial method that focuses on strengthening the training of low-frequency identifiers. The primary objective of VarGAN is to align the distribution of low-frequency tokens to their corresponding high-frequency counterparts within the representation space. This alignment is achieved through a Generative Adversarial Network (GAN) framework [9], wherein we regard the code representation model as the generator responsible for producing vectors from source code. Additionally, we employ a classifier referred to as the discriminator. The discriminator's role is to assess the encoding generated by the generator and determine whether the code contains low-frequency words. Conversely, the generator endeavors to deceive the discriminator, effectively making it unable to differentiate whether the input code comprises low-frequency words or not. This adversarial setup incentivizes the generator to produce more precise representations of low-frequency identifiers. In this way, the generated low- and high-frequency variable vector spaces can overlap as much as possible, thereby alleviating the above-mentioned problem highlighted in Figure 1.

To verify the effectiveness of our proposed method, we conducted experiments from three aspects: *Firstly*, we verify whether VarGAN can improve the representations of low-frequency identifiers in the vector space. Specifically, we examined the distribution of vectors before and after applying our method. We observed that our method leads to more uniform distribution of vectors for low-frequency identifiers. *Subsequently*, we evaluate the model's capability in identifier understanding using the identifier similarity scoring task. Evaluation results show that our proposed method can help the model better capture the semantic meaning of identifiers. *Finally*, we apply VarGAN to several downstream tasks, including similar identifier search, code summarization, and code clone detection. The results prove that adversarial training of low-frequency identifiers can boost the performance of CodeBERT, which affirms the practicality of our proposed method.

To summarize, our main contributions are as follows:

- We study the issue of imbalanced training of low- and high-frequency identifiers within code representation models.
- We propose a novel adversarial method to mitigate the gap between low- and high-frequency variable names in the vector space learned by code representation models.
- We conduct extensive experiments to evaluate the effectiveness of our approach. The results show that VarGAN achieves a new state-of-the-art performance on the identifier similarity scoring benchmark and also improves code representation models in other downstream tasks.

2 Background

2.1 Variable Name Representations

Variable name representation is a technology to encode variable identifiers into continuous vectors, such that identifiers have close relationships also share similar vectors [1]. These encoded vectors, or named embeddings, can be used for identifier correction and renaming to facilitate code analysis and manipulation.

A traditional way for variable representation is the utilization of Word2Vec [22] which assigns embeddings to each subtoken in the code and then trains a neural network to predict the target identifier given context embeddings. Word2Vec has the bag-of-words assumption and thus fails to generate context-aware vector representations for individual identifiers [37].

Recently, researchers have started utilizing pre-trained language models to obtain vector representations for source code tokens, including identifiers. Pre-trained models such as CodeBERT are built upon the Transformer architecture, utilizing a self-attention mechanism to allow a token to attend to other tokens in the same sequence [40]. Through training on a large code corpus, these pre-trained models have a better understanding of the identifier semantics in the broader context of the code.

However, directly utilizing pre-trained models to generate identifier representations often produces suboptimal outcomes. This is mainly caused by the existence of semantically similar yet distant identifiers. To tackle this problem, Chen et al. propose VarCLR [3], where the contrastive loss function is augmented to the identifier embeddings. This augmentation encourages the pre-trained model to learn representations that exhibit proximity for semantically similar variables and distance for dissimilar variables.

2.2 Generative Adversarial Networks

Generative Adversarial Network (GAN) is a machine learning technique aiming at generating high-quality data samples that resemble the training corpus [9]. A GAN consists of two neural networks competing with each other: a generator G and a discriminator D. G aims to generate data samples (such as images and texts) from random noise z,

namely $\hat{x} \sim G(z)$, while D is responsible for distinguishing between real and model generated samples. Given an input sample x, it predicts $p_{\text{real}} = D(x)$, the probability that the input is real.

During the training progresses, G and D alternatively improve their capabilities through optimizing a min-max objective:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim\mathcal{D}}[\log D(x)] + \mathbb{E}_{z\sim P(z)}[\log(1-D(G(z)))] \tag{1}$$

Upon reaching equilibrium, G will produce increasingly realistic samples, and D will encounter increasing difficulty in distinguishing between real and generated samples. The ultimate objective of GAN is to train a generator network G whose output samples are indistinguishable from real ones.

2.3 Downstream Tasks

Programming-related downstream tasks serve as a comprehensive means to evaluate the capabilities of code representation models. As our paper primarily focuses on encoder-only models, we study downstream tasks that are related to code comprehension, including:

Code Summary. A task to generate a natural language summary for a given code snippet [35]. A model generating high-quality code summaries means that it has a high capability of code comprehension.

Clone Detection. Given a code snippet and a set of candidate snippets as input, this task aims at identifying the top-K code snippets that are semantically similar to the input code snippet [26].

3 METHOD

In this section, we will introduce VarGAN, a novel adversarial method to mitigate the gap between low- and high-frequency identifiers.

3.1 Overview

We regard a PLM-based code encoder (e.g., CodeBERT) as a generator that produces accurate semantic vectors for a given code snippet. Our objective is to align the vectors of low-frequency identifiers to the corresponding high-frequency counterparts in the vector space. To this end, we accompany the generator with a discriminator that detects whether the input code of the generator contains low-frequency identifiers. The generator's objective is to produce vectors that the discriminator cannot distinguish, thereby overlapping the generated vectors of both low- and high-frequency identifiers as much as possible in the vector space.

Our method comprises three components, as illustrated in Figure 2. Firstly, we partition the training code corpus into low- and high-frequency subsets. Secondly, we harness an adversarial training process to regularize the vector space of low-frequency identifiers to its high-frequency counterparts. Lastly, we integrate our method with other pre-training tasks of the PLM encoder to ensure the vectors preserve the semantics in the code context. We will present each of these components in detail in the following sections.

3.2 Partitioning Code with Identifier Frequency

Our method involves a classifier that discriminates whether a given code snippet contains low-frequency variable names. To train this classifier, we need both positive (i.e., code with low-frequency identifiers) and negative (i.e., code without low-frequency identifiers) samples. We accomplish this by partitioning the original pre-training corpus (i.e., CodeSearchNet) for the PLM into two subsets. Let $\mathcal{C} = \{x_1,...,x_N\}$ denote the code corpus for pre-training the PLM. For each code snippet $x_i \in \mathcal{C}$, we assign a label $y_i \in \{0,1\}$. $y_i = 1$ if x_i contains low-frequency identifiers and 0 otherwise. This yields a labeled dataset of $\tilde{C} = \{(x_1,y_1),...,(x_N,y_N)\}$.

This raises the question of how to define low-frequency identifiers. In our approach, an identifier is low-frequency only when the occurrence of it or any of its subtokens falls below threshold α. α is the median value of the frequency of subtokens of all identifiers inside the code corpus. Therefore, it creates a rough balance between the positive and negative samples for training the discriminator. We count the frequency of all identifiers from the CodeSearchNet code corpus [13]. Observed that camel case and underscore are commonly used for combining variable names [1], we split identifiers using the *safe_simple_split* method from the Spiral package[1] and count the subtokens instead.

3.3 Code Representation Learning (Generator)

We adopt the popular PLM-based code representation model, CodeBERT, as a generator that produces feature vectors for a code snippet. Given a code snippet $x = w_1,...,w_n$ with n subwords, the PLM generator produces a sequence of hidden vectors

$$\mathbf{h}_1,...,\mathbf{h}_n = \text{CodeBERT}(w_1,...,w_n)$$

where $\mathbf{h}_i \in \mathbb{R}^d$ denotes the encoding vector for token i.

We then perform average pooling on the vector sequence to obtain the vector representation of the entire code snippet:

$$\mathbf{c} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{h}_i. \tag{2}$$

Therefore vector representation of the code snippet $\mathbf{c}$ has the same dimension as each subword vector $\mathbf{h}_i$.

3.4 Rare Identifier Detection (Discriminator)

With the generator, the infrequent identifiers in x will be reflected in the encoded vector $\mathbf{c}$. Therefore, we create an additional discriminator to regularize code vectors produced by CodeBERT. The discriminator is a fully connected neural network that performs binary classification on the code vectors produced by the generator. Specifically, it takes code vector $\mathbf{c}$ as input and outputs a confidence score between 0 and 1 indicating how likely the input code x to the generator contains low-frequency identifiers.

$$D(x) = \text{sigmoid}(\mathbf{W}\mathbf{c}) \tag{3}$$

where $\mathbf{W}$ denotes the training parameters.

1. https://github.com/casics/spiral

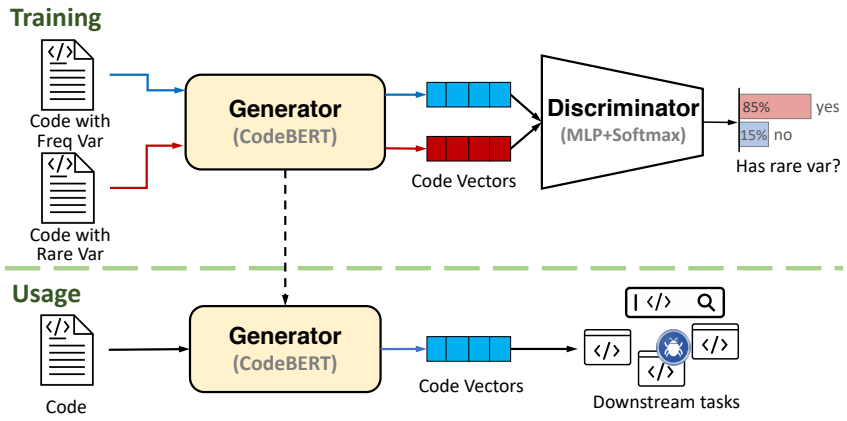


Fig. 2: The overall framework of VarGAN

3.5 Adversarial Training

Throughout the training process, samples were drawn from the training corpus and subsequently used to compute the loss functions by feeding them through the generator and discriminator. The training objective of the discriminator is to accurately identify whether the code contains low-frequency identifiers. Therefore, it aims to minimize the binary cross-entropy loss:

$$\mathcal{L}_D = -\frac{1}{N}\sum_{i=1}^{N}(y_i \log D(x_i) + (1-y_i)\log(1-D(x_i))) \quad (4)$$

where N represents the number of training samples.

The objective of the generator G is to deceive the discriminator D, namely, maximize the error rate of the discriminator D. Therefore, the loss function of the generator is the negative log-likelihood of the discriminator's misclassification probability:

$$\mathcal{L}_G = -\frac{1}{N}\sum_{i=1}^{N}((1-y_i) \log D(x_i) + y_i \log(1-D(x_i))) \quad (5)$$

Overall, the adversarial training aims to optimize the min-max objective:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim\mathcal{C}_h}[\log D(x)] + \mathbb{E}_{x\sim\mathcal{C}_l}[\log(1-D(x))] \quad (6)$$

where $\mathcal{C}_h$ and $\mathcal{C}_l$ denote code corpora that contain high- and low-frequency identifiers, respectively.

To accomplish this min-max game, in the training process, we alternatively update gradients for the discriminator and generator toward their objectives. It is also imperative to note that throughout the training process, G must not be excessively trained without updating D, otherwise the generator may "take shortcuts" by generating certain singular values that can confuse the discriminator, but this comes at the expense of losing the ability to generate diverse samples. To mitigate this challenge, we introduced a discriminator update step (d_iter) in the iteration process. Specifically, the discriminator underwent multiple iterations prior to a single iteration of the generator, thus facilitating the acquisition of more comprehensive learning by the discriminator. The whole process is outlined in Algorithm 1.

Through adversarial training, we anticipate the distribution of low- and high-frequency identifiers to be more uniform so that the discriminator cannot detect low-frequency identifiers from the representation vectors. Figure 3 illustrates the idea of vector space alignment by adversarial training, where G_0 and D_0 denote the initial generator and discriminator, while G_K and D_K represent the final model attained upon convergence of the training process. Initially, there exists a notable discrepancy between the distributions of low- and high-frequency identifiers (as can be seen from the upper part of the figure). However, as adversarial training proceeds, there is a consequential shift observed in the identifier vectors. Gradually, the vector distributions of low- and high-frequency identifiers begin to overlap with each other, as depicted in the bottom part of the figure, thereby alleviating the situation where semantically similar identifiers are far apart in the vector space due to frequency differences.

3.6 Preserving the Original Semantics

As adversarial training only encourages PLMs to generate variable vectors that cannot be distinguished by the discriminator, the learned vectors could deviate from the original semantics during pre-training. To preserve the semantics of variables in the generated vectors, we integrate the adversarial training process with the original MLM objective of CodeBERT. Since we only concern identifiers, we chose the Masked Identifier Prediction (MIP) instead, which is widely used in code PLMs [28], [42]. Specifically, we randomly mask 50% of the identifiers from the input code x and ask the pre-trained CodeBERT model to predict the subwords. Let $\hat{x}$ denote the masked code snippet of x. $p_\theta(.)$ denotes a multi-layer perceptron that maps the hidden vector of CodeBERT to the vocabulary space. The MIP task aims to minimize the cross-entropy loss:

$$L_{sc}(\theta) = \sum_{i\in n} -\log p_\theta(x_i|\text{CodeBERT}(\hat{x})) \quad (7)$$

We integrate the MIP loss with the adversarial training loss of the generator and jointly perform backpropagation.

3.7 Training

We treat VarGAN as a pre-training method of a PLM. We selected CodeBERT as our backbone PLM and continually pre-train it using VarGAN.

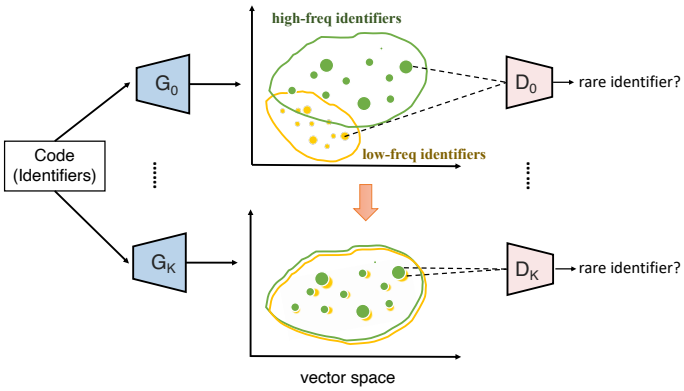


Fig. 3: Illustration of vector space alignment by VarGAN

Algorithm 1: Adversarial Training of VarGAN

Input: $\tilde{\mathcal{C}}$={(x_1,y_1),...,(x_N,y_N)}: A code corpus with labels of low-frequency identifiers;
G: A PLM for code representation (e.g., CodeBERT);
D: A discriminator to detect low-frequency identifier in **G**;
d_iter: discriminator update steps;
Output: G: The optimized generator.

while *not converge* **do**
  for $i \in$ *steps* **do**
    ▷ traverse all batches in data loader
    Select (x, y) from $\tilde{\mathcal{C}}$;
    c = G(x); ▷ generate vectors for code
    $p(x)$ = D(**c**); ▷ detect low-frequency identifiers
    Compute L_d using Equation 4;
    Compute L_g using Equation 5;
    $\tilde{x}$ = mask_identifier(x);
    Compute L_{sc} using Equation 7 ;
    Update **D** with L_d;
    if i % d_iter == 0: **then**
      Update **G** with $L_{sc} + L_g$;
    end
  end
end

Dataset: We train CodeBERT using the Java subset of CodeSearchNet [13], which consists of 454,451 code snippets. We employ tree_sitter [2] to parse the code snippets and extract the identifiers. We set the threshold of low-frequency identifiers to 301 which yields a rough balance between positive and negative training samples.

Setting: The discriminator used in our approach is a multi-layer perceptron (MLP) with two layers, with the input dimension being the same as the output dimension of the PLM. Since the discriminator is randomly initialized, the generator can easily produce vectors to deceive the discriminator at the early stage. To allow the generator to receive sufficient training, we set a small learning rate of 5e-7 for the generator and a larger learning rate of 5e-5 for the discriminator. The hyperparameter d_{iter} in Algorithm 1 is set to 2. To ensure consistency with downstream tasks, we extract the final-layer hidden states from CodeBERT as the outcome of the generator and utilize them for adversarial training. The batch size is set to 32, and the training is conducted for ten epochs.

2. https://github.com/tree-sitter/tree-sitter

4 Evaluation

In this section, we conduct experiments to validate the effectiveness of the proposed approach. Overall, we address the following research questions:

- **RQ1: Does VarGAN lead to a more uniform distribution of low- and high-frequency identifier vectors?**
  This RQ aims to examine the impact of VarGAN on the regularization of the vector space learned by CodeBERT. Answering this RQ enables us to understand the fundamental reasons for the effectiveness of VarGAN.
- **RQ2: How effective is VarGAN in variable semantic representation?**
  As the primary goal of this paper, we showcase the effectiveness of our approach in capturing the semantics of variable names. Following a recent variable representation study VarCLR [3], we gauge the model's understanding of variable semantics by assessing its performance in variable-relevant downstream tasks such as variable similarity scoring and variable search.
- **RQ3: Does VarGAN enhance code representation models on downstream tasks?**
  In addition to assessing the efficacy of capturing variable semantics, we are interested in evaluating how well VarGAN captures code-level semantics. To investigate this aspect, we employ our proposed approach as a pre-training task for code representation models and assess their performance on downstream tasks.

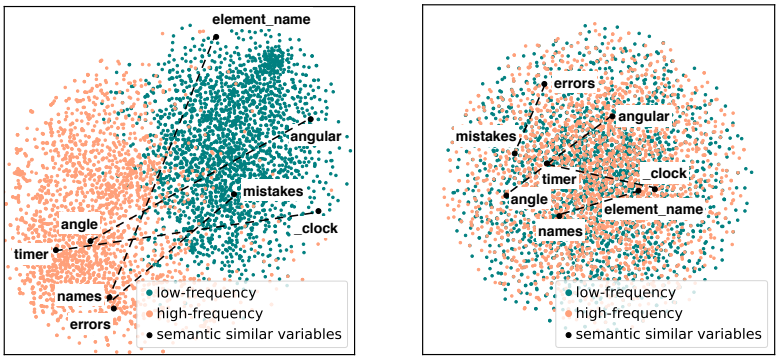
4.1 The Spatial Distribution of Identifier Vectors (RQ1)

In this section, we study the effect of VarGAN in regularizing the vector space learned by CodeBERT. We start with a qualitative analysis demonstrating the change of distributions in the vector space before and after incorporating VarGAN, followed by a quantitative measuring of the variable distances through the adversarial learning process.

4.1.1 Qualitative Analysis

To investigate the impact of our method on the distribution of variable vectors, we visualized the vector space of variables learned by CodeBERT before and after adding VarGAN during pre-training. We pre-train CodeBERT using the methodology in Section 3.7. Then, we select the 5000 most frequent and rare identifiers respectively from the vocabulary. We feed each identifier into CodeBERT and obtain its vector using the hidden state from the last layer. We use the T-SNE [38] to reduce the dimensionality of the vectors and plot them in the 2D space. The results are rendered in Figure 4.

In the absence of applying VarGAN (a), the identifier vectors produced by CodeBERT exhibit a separate distribution. While the vectors of high-frequency identifiers span a wider stretched space, low-frequency ones were concentrated near the origin. This is caused by the different frequencies between identifiers in the corpus. Word vectors are



(a) Vanilla CodeBERT (b) CodeBERT with VarGAN

Fig. 4: T-SNE visualization of variable embeddings learned by CodeBERT and VarGAN

typically initialized around the origin with small variances, but high-frequency identifiers have more opportunity for training and moving away from the origin compared to those of low-frequency ones [8].

Such a separate distribution further causes the semantic discrepancy problem, that is, the distance between vectors does not reflect the variable similarity. For example, both $timer$ and $\_clock$ mean a utility about time, though developers prefer using $timer$ (with a high frequency) than $\_clock$ (with a low frequency). We observe that their vectors are far apart in the space learned by CodeBERT. Similarly, while $errors$ and $mistakes$ possess comparable semantic meanings, their significant disparity in word frequency results in a substantial distance between the two variables.

Upon applying VarGAN (b), we observed a more uniform distribution of identifier vectors. The separation between low- and high-frequency identifiers decreased, indicating that the identifier vectors were more uniformly distributed. We attribute this improvement to our adversarial training approach, which prompts CodeBERT to generate vectors for low- and high-frequency identifiers that are challenging to distinguish for the discriminator. Furthermore, we notice a significant reduction in the distance between previously semantically related variables, such as $timer$ and $\_clock$, or $errors$ and $mistakes$. This decrease in distance suggests that the model has developed a more refined understanding of variable semantics by incorporating our method.

The analysis above highlights that the effectiveness of VarGAN primarily arises from its adversarial training approach, resulting in a more balanced distribution of vectors. By countering the discriminator's judgments, the model naturally avoids segregating low- and high-frequency variables into separate clusters. Consequently, the distances in the vector space can more accurately reflect the correlation between variables, irrespective of their frequency. The improved performance of the model in downstream tasks is largely attributed to this characteristic.

4.1.2 Quantitative Analysis

To verify the qualitative findings, we conducted a quantitative analysis by calculating the average pairwise cosine similarity within each group of variables and evaluating the cross-group similarity. This approach enables us to quantitatively assess the distributions of the two groups of variables. The results are presented in Table 1.

TABLE 1: Average pairwise cosine similarities for different groups of variables

Model	Within Group		Cross Group
	High-freq.	Low-freq.	
CodeBERT	0.3998	0.5304	0.4357
CodeBERT + Ours	0.5845 (↑)	0.5842 (↑)	0.5843 (↑)

In the original CodeBERT, we observed that the high-frequency group exhibits smaller pairwise similarities compared to the low-frequency group. This suggests that high-frequency variables are more spread out in the vector space, while low-frequency variables are more compressed. Additionally, the cross-group similarity is relatively low, indicating a clear separation between the two groups of variables. These observations underscore the presence of the previously noted imbalanced training issue, which can have adverse effects on downstream tasks such as identifier similarity search.

Upon adversarial training, we observed a significant increase in pairwise cosine similarities for both groups. Additionally, the average similarities for the two groups become comparable, indicating a more balanced distribution of variables. Moreover, the cross-group similarity shows a clear increase, implying that our approach effectively aligns low-frequency identifiers with their high-frequency counterparts. As a result, a more uniform distribution is achieved across both groups of variables, indicating its effectiveness in alleviating imbalanced identifier training.

Answer to RQ1: VarGAN empowers CodeBERT to stretch the distribution of identifier vectors and produce more uniform distributions of low- and high-frequency identifier vectors, indicating its effectiveness in alleviating imbalanced identifier training.

4.2 Effectiveness in Variable Semantic Representation (RQ2)

We subsequently showcase the effectiveness of our approach in capturing the semantics of variable names. Following a recent variable representation study VarCLR [3],

we conduct our evaluation through two tasks, *variable similarity scoring* and *variable similarity search*.

It is important to note that VarGAN differs from VarCLR in terms of task specification. While VarCLR represents only single variables, VarGAN aims to encode the entire code. To accommodate this task and ensure a fair comparison, we feed the generator with only individual variables. The objective of the discriminator is modified to determine whether the input variable has a low frequency in the training set. We also exclude the MIP task in the adversarial training process. We refer to the variant of our method as *VarGAN-S*. VarCLR-S is not comparable to VarCLR directly. Consequently, we integrate our approach into VarCLR's framework and evaluate its performance in comparison to VarCLR.

4.2.1 Variable similarity scoring

The goal of this task is to verify whether the (cosine) similarity scores between variables computed using model-generated vectors are correlated with the human-assigned ground-truth scores. We measure the correlation between the two parallel scores using Spearman's rank correlation coefficient [30]. The coefficient falls in the range [-1, 1]. A higher coefficient indicates more accurate semantics captured by the model.

Dataset: Following VarCLR, we use the GitHubRenames dataset [3] for training the variable representation model and the IdBench scoring benchmark [41] for testing.

GitHubRenames comprises 66,855 pairs of similar variables, each containing a variable name before and after a renaming commit. We build a vocabulary based on the variable frequencies in this dataset and label whether each variable has a low frequency. We set the threshold α to 63 in this dataset.

IdBench is a benchmark that is widely used for evaluating variable semantic representations [3]. The benchmark consists of 176 pairs of similar variables that were manually labeled by 500 software developers. Each pair is labeled with a similarity and relatedness score, where similarity refers to the degree of shared meaning, allowing for substitution without altering the overall meaning; and relatedness captures the level of association, encompassing diverse potential relationships. The benchmark is grouped into three subsets (i.e., small, medium, and large) with varying data cleaning thresholds.

Settings: To demonstrate the versatility of our approach, we applied VarGAN-S to three distinct pre-trained models: RoBERTa [19], trained on natural language; CodeBERT [6], trained on code corpora; and GraphCodeBERT [10], integrated code structure information. We apply VarGAN-S to each PLM and compare the performance to the original one. We also build VarCLR based on the PLMs (denoted as VarCLR-PLM) and compare the performance with and without VarGAN-S.

We adhered to the training parameters as stipulated in the original paper, with the sole exception of configuring a learning rate of $2e$-5 for the newly introduced discriminator.

Results: Table 2 summarizes the experimental results. Overall, VarGAN-S exhibits a clear performance enhancement in each comparison pair (bolded in Table 2). The pre-trained model achieves low correlations in terms of similarity (0.12-0.42) and relatedness (0.25-0.68) across all test sets. After applying VarGAN-S, the similarity score increases to 0.22-0.43 and relatedness to 0.37-0.70. VarGAN-S also demonstrates better performance when applied to VarCLR. The similarity score increases to 0.43-0.57 and relatedness to 0.74-0.83. VarGAN-S further enhances the improvement by around 8%.

TABLE 2: Spearman's rank correlation with IdBench in terms of similarity (a) and relatedness (b)

(a) Similarity scores

Model	Small	Medium	Large
RoBERTa	0.18	0.16	0.21
+VarGAN-S	**0.23**	**0.22**	**0.25**
CodeBERT	0.13	0.13	0.12
+VarGAN-S	**0.31**	**0.24**	**0.26**
GraphCodeBERT	0.41	0.42	0.40
+VarGAN-S	**0.43**	**0.43**	**0.42**
VarCLR-RoBERTa	0.45	0.41	0.42
+VarGAN-S	**0.48**	**0.43**	**0.43**
VarCLR-CodeBERT	0.53	0.53	0.51
+VarGAN-S	**0.57**	**0.55**	**0.55**
VarCLR-GraphCodeBERT	0.54	0.54	0.53
+VarGAN-S	**0.57**	**0.56**	**0.55**

(b) Relatedness scores

Model	Small	Medium	Large
RoBERTa	0.36	0.36	0.38
+VarGAN-S	**0.37**	**0.40**	**0.40**
CodeBERT	0.25	0.28	0.29
+VarGAN-S	**0.40**	**0.45**	**0.49**
GraphCodeBERT	0.68	0.67	0.67
+VarGAN-S	**0.69**	**0.70**	**0.70**
VarCLR-RoBERTa	0.69	0.70	0.70
+VarGAN-S	**0.74**	**0.74**	**0.75**
VarCLR-CodeBERT	0.79	0.79	0.80
+VarGAN-S	**0.81**	**0.81**	**0.82**
VarCLR-GraphCodeBERT	0.84	0.82	**0.83**
+VarGAN-S	**0.86**	**0.83**	**0.83**

We notice that a single VarGAN-S exhibits lower performance compared to VarCLR when applied to a PLM. This discrepancy arises from VarCLR being trained on parallel variables that share semantic similarity, a condition not required by VarGAN-S. Furthermore, VarCLR takes only variables as input, whereas the original VarGAN is designed to encode the entire code. To accommodate this task difference, VarGAN-S omits the semantic preservation step from VarGAN, leading to a decrease in performance. This observation also indicates that simply adversarial training without semantic preservation is insufficient to regularize the variable representations.

To gain further insights into the results, we conducted an analysis of the effects of using different proportions of training data on the model's performance. As depicted in Figure 5, our method demonstrates a more significant advantage in scenarios with limited training data. Specifically, when only 10% of the training data is available, VarGAN-S enhances the baseline models by approximately 30%. The rationale behind this observation is straightforward: in low-resource settings, the challenge of insufficient training for low-frequency variables becomes more prominent. With a larger amount of training data, low-frequency variables have more opportunities for training and exploitation. Con-

TABLE 3: Spearman's rank correlation (similarity) with IdBench for each layer of CodeBERT. 0 stands for the first (embedding) layer while 12 represents the last layer.

	Small		Medium		Large	
Layer	**Orig.**	**VarGAN-S**	**Orig.**	**VarGAN-S**	**Orig.**	**VarGAN-S**
12	0.13	0.31	0.13	0.24	0.12	0.26
11	0.14	0.32	0.14	0.25	0.12	0.26
10	0.15	0.29	0.16	0.23	0.14	0.23
9	0.20	0.23	0.18	0.20	0.17	0.19
8	0.25	0.21	0.24	0.19	0.22	0.17
7	0.27	0.26	0.27	0.27	0.25	0.23
6	0.27	0.25	0.27	0.26	0.24	0.22
5	0.27	0.24	0.27	0.25	0.24	0.21
4	0.28	0.26	0.27	0.26	0.25	0.23
3	0.26	0.25	0.28	0.27	0.26	0.25
2	0.28	0.26	0.30	0.29	0.27	0.26
1	0.28	0.28	0.30	0.30	0.27	0.27
0	0.24	0.24	0.26	0.26	0.21	0.21

sequently, the problem of disparate distribution is alleviated. These results further validate our initial motivation regarding the inadequacy of training for low-frequency variables.

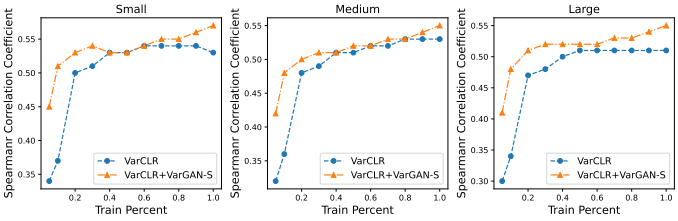


Fig. 5: Spearman's rank correlation (similarity) under different training data percent

Case Study. We take CodeBERT as an example to illustrate how VarGAN-S enables CodeBERT to achieve almost twice the higher similarity/relatedness score. Table 3 shows Spearman's correlation with IdBench in terms of similarity using the vectors from each layer of CodeBERT. As the results indicate, the improvement of VarGAN-S becomes more pronounced using later layers, indicating that the vectors obtained from the earlier layers are more uniform. This could be caused by two impacts: 1) the last layer is used for training, and thus the optimal vector naturally appears in this layer; 2) the first few layers have a similar distribution of low- and high-frequency variables, while they are closer to the embedding layer and thus are easier to find low-frequency subwords in encoding vectors.

4.2.2 *Variable Similarity Search*

Variable similarity search is a task to find the most similarity variables from a candidate pool [3]. Given a pair of variables recognized as similar, along with a substantial set of candidate variables, the objective is to compute the cosine similarity between the input variable and all the variables within the candidate set. This similarity calculation enables the selection of the top-k variables that possess the greatest resemblance. Subsequently, a verification process is conducted to ascertain the presence of the previously identified similar variables within the selected subset of k variables [3].

Dataset: We adhere to the settings established in VarCLR. The GitHubRenames dataset was used to train the model. Variable pairs within the IdBench-large dataset possessing a similarity score greater than 0.4 are selected as similar pairs to test the model. Moreover, the competing variable data aligns with VarCLR, encompassing a total of 209,388 candidate variables.

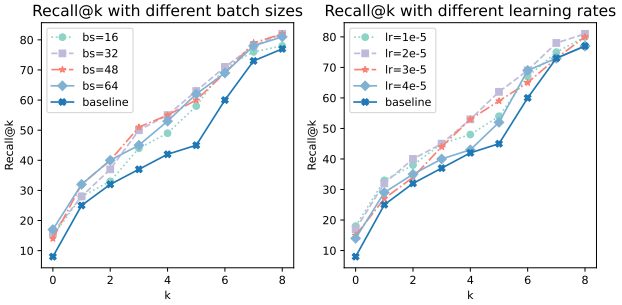


Fig. 6: Results of variable similarity search under different settings

Results: The experimental results are depicted in Table 4. We measure the performance using recall and compare VarGAN-S with VarCLR, the previous state-of-the-art method in this task. The findings demonstrate a substantial superiority of our method over VarCLR, particularly evident when considering a small value of k. Notably, our method attains a new record recall of 17%, almost doubling the score achieved by VarCLR (8%).

Robustness Analysis: To demonstrate the robustness of our approach, we conducted additional experiments using different hyper-parameters in conjunction with external methods. These experiments involved variations in batch size, as well as utilizing a shared learning rate for both the generator and discriminator. It can be observed that our approach consistently achieves superior performance compared to the baseline method VarCLR.

Case Study. To further analyze the advantages of VarGAN-S in variable semantic representation, we present the top-5 most similar words of input variables found by VarCLR and the inclusion of VarGAN-S in Table 5. Notably, the integration of VarGAN-S facilitates a better acquisition of semantic information and enables the exploration of a more diverse vocabulary, in contrast to VarCLR, which primarily relies on visual similarities between word morphology. For instance, in many contexts, "cols" serves as a shortened form of "columns" and carries the same semantic meaning. While VarGAN-S successfully captures such an association, VarCLR only finds variables containing the exact word "column". Furthermore, VarGAN-S captures the association between "user" and "person", yet VarCLR only finds words containing "user". This highlights the advantage of VarGAN-S in uncovering meaningful linguistic relationships beyond surface-level similarities.

Nevertheless, there are still some cases where our approach exhibits suboptimal outcomes. It may associate variables with others that are contextually relevant but lack the exact semantic equivalence. As an example, VarGAN-S may link "inputs" to "inputNodes" and link "equals" to "equalsDate", highlighting a potential limitation in capturing precise semantic meanings in certain scenarios. Efforts to

TABLE 4: Comparing VarCLR and VarGAN-S on the variable similarity search task

Model	Recall@1	Recall@5	Recall@10	Recall@25	Recall@50	Recall@100	Recall@250	Recall@500	Recall@1000
VarCLR	8.0	25.0	33.0	37.0	42.0	45.0	60.0	73.0	77.0
+VarGAN-S	17.0	32.0	40.0	45.0	53.0	62.0	69.0	78.0	81.0

TABLE 5: Top-5 similar variables found by VarCLR and VarGAN-S in the variable similarity search task

Input	Reference	Method	Top-5 Similar Variables
user	person	VarCLR	$user, tuser, userid, userId, useraccount
		+VarGAN-S	author, userid, person, userId, username
columns	cols	VarCLR	sColumns, columnsArray, $columns, columnFields, rowColumns
		+VarGAN-S	columnFields, propertys, cols, columnsItems, relationFields
callback	cb	VarCLR	callbackfn, callbackFn, callbackAsync, fCallback, callbackify
		+VarGAN-S	cb, callbackfn, handler, callbackFn, callbackname
username	userid	VarCLR	userUsername, ssoUsername, $username, usernameInstance, accountUsername
		+VarGAN-S	userName, userid, loginname, $username, user
items	files	VarCLR	sitems, itemsList, eItems, mItems, itemsArray
		+VarGAN-S	files, values, lines, results, models
utils	util	VarCLR	utilities, tilsUtils, esutils, libutils, helperUtils
		+VarGAN-S	util, tools, math, forge, runner
inputs	resources	VarCLR	input, $inputs, inputSources, inputsEl, sInput
		+VarGAN-S	$inputs, inputNodes, inputList, fieldInputs, inputsEl
equals	same	VarCLR	equalsFn, equalsF, notEquals, $equals, isEq
		+VarGAN-S	equalsFn, equalsF, isEquals, $equals, equalsDate

address and further refine these occurrences could enhance the overall accuracy and reliability of our method.

Answer to RQ2: Adversarial training enables PLMs to better capture variable similarities and relatedness, demonstrating its ability in learning variable semantic representations.

4.3 Effectiveness in Downstream Tasks (RQ3)

In this section, we present the effectiveness of our proposed method, VarGAN, by utilizing it as a pre-training task and subsequently fine-tuning it on downstream tasks as introduced in Section 2.3. Specifically, we evaluate the performance of VarGAN on two downstream tasks: code summarization and code clone detection. These tasks are chosen to showcase the effectiveness of VarGAN in capturing code-level semantics.

4.3.1 Code Summary

Setup: We evaluate the performance of VarGAN in Java code summarization. As CodeBERT is an encoder-only model, we append it with an additional Transformer decoder to accommodate the sequence-to-sequence task. Then, we fine-tune the models pre-trained with VarGAN on a parallel PL-NL dataset. Following CodeBERT, we use the data and scripts provided in CodeXGLUE [21]. We set the learning rate to $6e$-5 and the batch size to 32. We measure the performance using the BLEU score [24] and compare the result with that of the vanilla CodeBERT. We do not include VarCLR as a baseline since it is designed to encode single variables while our approach aims to encode the entire code. Besides CodeBERT, we also apply our approach to GraphCodeBERT [10], a follow-up code representation model that also has an encoder-only architecture. We did not consider other generative PLMs (e.g., CodeT5) which are beyond the primary scope of this paper. Besides, generative PLMs typically involve seq2seq pretraining objectives, which are incompatible with VarGAN at present.

Results: The results are presented in Table 6. As observed, our proposed method increases the BLEU scores by 3% and 2% compared to CodeBERT and GraphCodeBERT, respectively. The results indicate that regularizing low-frequency identifiers with adversarial training enhances code representation models in code comprehension.

We also evaluate the contribution of each training objective by ablating either MIP or GAN from VarGAN. As the results show, either GAN or MIP is useful for code semantic learning. Moreover, the inclusion of MIP further enhances the performance.

4.3.2 Clone Detection

Setup: We experiment with the POJ-104 [23] benchmark, which contains 104 programming problems from an open judge (OJ) system [3]. Each problem has 500 corresponding c/c++ solutions that are considered semantically similar code snippets. We evaluate the performance using the MAP@R metric, which is defined as the mean of average precision scores. Each average precision score is evaluated for retrieving R most similar samples given a query. We set the maximum value of R to 499 since each code snippet has at most 499 similar snippets. We set the learning rate to $2.4e$-5 and the batch size to 16. We apply our method to CodeBERT and GraphCodeBERT and compare the results with the vanilla models. We do not include VarCLR as a baseline since it is designed to encode single variables while our approach aims to encode the entire code. We reproduce the results of baseline models using the default settings and scripts provided on the CodeXGLUE website[4].

Results: The experimental results are presented in Table 6. It can be observed that after applying our method to CodeBERT and GraphCodeBERT, their MAP scores in the

3. http://programming.grids.cn
4. https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Clone-detection-POJ-104

clone detection task have improved by 0.8 and 3.5, respectively. These results further demonstrate the effectiveness of VarGAN in capturing code semantics.

We also conducted an ablation experiment on CodeBERT and GraphCodeBERT with solely MIP and GAN training. We observe that utilizing MIP only leads to a decrease in performance whereas sole reliance on GAN training continues to significantly enhance overall performance. This observation underscores the efficacy of adversarial training when employed as an independent task, while MIP may benefit more from being combined with other tasks.

TABLE 6: Performance of code summarization (Java) and clone detection (C/C++) (CS: code summarization, CD: clone detection)

Task	CS	CD
Metric	BLEU	MAP@R
RoBERTa	16.47	76.7
CodeBERT (paper)	17.65	82.7
CodeBERT (reproduce)	18.58	86.8
CodeBERT+VarGAN	**19.11**	**87.6**
-w/o MIP	18.73	87.3
-w/o GAN	18.68	86.3
GraphCodeBERT	18.78	86.7
GraphCodeBERT+VarGAN	**19.15**	**90.2**
-w/o MIP	19.01	88.4
-w/o GAN	18.80	87.6

[1] We do not include VarCLR as a baseline since it is designed to encode single variables while our approach aims to encode the entire code.

Answer to RQ3: Through adversarial training, VarGAN enhances the performance of PLMs in downstream tasks, due to its superb specialty in capturing code-level semantics.

5 Discussion

5.1 Limitations

On one hand, VarGAN adopts GAN to improve the quality of variable semantic representations, which inevitably increases the training time. As GAN requires the generator and the discriminator to be trained iteratively, it introduces 20% overhead compared to only training a generator. In future work, we will explore optimization solutions to reduce training costs. One direction is integrating adapter-based methods. Instead of updating all the model parameters, they selectively adjust a small portion of them, which allows applying our VarGAN to larger models. Another direction is to follow the latest adversarial training study to reduce training costs and improve model accuracy.

On the other hand, as VarGAN primarily focuses on adjusting the distribution of encoding vectors, it is limited in its applicability solely to encoder-only PLMs. Specifically, the absence of an encoding space in a decoder-only architecture precludes the application of this approach. To address this constraint, future research may pursue the adaptation of VarGAN for generative models and Large Language Models (LLMs), which would broaden its scope of potential applications. By enabling its use in a wider range of tasks, this extension may further enhance the versatility and utility of VarGAN.

5.2 Road Ahead

Our method is only built and verified in encoder-only code representation models. Hence, it remains to investigate the adaptation to code models with other architectures, particularly the decoder-only large language models. In the future, we will design new methodologies to adapt VarGAN to decoder-only models. For example, we can use the output of the last Transformer block before the output layer as the encoding vectors for each output token. An adversarial training approach can be employed by introducing a classifier to distinguish between low- and high-frequency variables. By incorporating this additional loss, we can enhance the decoder-only models' capability to produce more accurate and contextually meaningful output, particularly when dealing with low-frequency words.

Additionally, besides the evaluation in this paper, we will further investigate the theoretical foundation of our approach. This helps to verify that our method is effective as well as sound from a theoretical perspective. We also plan to extend VarGAN to other code-related tasks, including code translation and alignment between code and natural language.

6 Related Work

6.1 Identifier Representation Learning

Identifiers play a pivotal place in source code and encapsulate important semantic information [25]. Identifier representation learning has been a widely-studied in code intelligent tasks, including identifier renaming [15], [39], [18], function name prediction [1], [17], and code clone detection [7]. Consequently, the area of identifier representation learning has gained substantial attention within the field of software engineering [41], [3], [20].

The conventional approach to obtaining identifier representation involves generating vectors using Word2Vec [22]. As a representative work, Chirkova et al. [4] adopt dynamic embeddings to capture variable representations not only from its name but also its specific role in the source code. More recently, researchers have employed pre-trained language models for code representation. CodeBERT [6] pioneers the research on PLM for code. It adapts RoBERTa [19] to code datasets. As a follow-up work, GraphCodeBERT [10] further abstracts code into graphs to accurately capture dependencies and semantic structures present within the code syntax. Regarding the importance of identifiers in code semantics, researchers in pre-trained language models designed identifier-related tasks as pre-training objectives, with the aim of strengthening the model's understanding of identifiers and enhancing the comprehension of entire code segments [28], [42], [2]. Our work builds upon these previous efforts and further advances the understanding and comprehension capabilities of identifiers. Additionally, our work introduces adversarial learning in a unique way to overcome the bias caused by identifier frequency.

To our best knowledge, there is only one previous work that is closely related to ours. Chen et al. proposed VarCLR [3], which represents the state-of-the-art approach for learning identifier representations. VarCLR tackles the challenges of embedding semantically similar but distant

identifiers by leveraging contrastive learning on a parallel identifier dataset, which yields considerable advancements over prior methods in the IdBench benchmark. VarGAN differs from VarCLR significantly in objectives and task specifications. Unlike our work which encodes the entire code, VarCLR merely encodes single identifiers. Moreover, VarCLR is a supervised approach that depends on parallel variable pairs while VarGAN is an unsupervised pre-training objective that is independent of parallel identifiers. Additionally, VarGAN tackles the imbalanced training problem for low-frequency identifiers through adversarial learning. Thus it is capable of finding identifiers that are more broadly relevant.

Another critical issue that arises in the representation of identifiers pertains to out-of-vocabulary (OOV) problems, which stem from the fact that software developers have the freedom to create any identifiers [14]. Various approaches have been proposed to tackle this challenge, including anonymization of low-frequency identifiers [5], splitting identifiers into subtokens [12], [20] and the use of byte-pair encoding (BPE) tokenization [14], [47]. As a representative work, Shi et al. [32] present a novel hybrid strategy that combines identifier splitting and the BPE algorithm to further optimize the performance of open-vocabulary models. Diverging from prior work, our VarGAN enhances the quality of identifier representation from a new perspective. It leverages adversarial networks to address the imbalanced training of low- and high-frequency identifiers by pre-trained language models (PLMs), and thus further improve the performance of prior work.

6.2 Representation Learning for Low-frequency Words

The imbalanced training between low- and high-frequency words is also a critical problem in the NLP domain, which causes the anisotropy and uneven distribution of embedding vectors by PLMs [16], [44]. Researchers have proposed numerous methods to tackle this problem [16], [29].

One line of work [16], [36] adjusts the distribution through vector space transformation. BERT-flow [16] employs a transformation to convert BERT-encoded sentence representations into a more isotropic and uniform distributed space. BERT-whitening [36] proposes a whitening operation that directly corrects the covariance matrix of local vectors.

Another line of work enhances the training for low-frequency words. BERTRAM [29] introduces an attention mimic mechanism to generate encoding vectors for low-frequency words that are more consistent with high-frequency words. FRAGE [8] improves the gap between the distribution of low- and high-frequency words using a GAN-based approach. DictBERT [44] incorporates an additional dictionary to enhance the model's understanding of low-frequency words. FCL [45] proposes a contrastive learning method that considers word frequency, enabling the model to pay more attention to low-frequency words.

Our VarGAN bears a resemblance to FRAGE [8], as both employ GAN in their architecture. However, while FRAGE focuses on addressing the natural language word embedding in traditional LSTMs [11], VarGAN specifically tackles rare variables (which are more pronounced challenges) in code PLMs. Incorporating adversarial training into code PLMs proves to be a nontrivial task due to the nuanced differences in pre-training pipelines compared to traditional LSTMs. Code PLMs involve multiple pre-training objectives, which poses special challenges such as how to seamlessly integrate adversarial training while preserving semantics during pre-training. To address these issues, our method proposes a new pipeline and training scheme (e.g., the semantic preservation step) that better coordinates adversarial training with pre-training objectives. Ablation results in the experiments also affirm that VarGAN with the new pipeline is more effective than simply applying adversarial training to PLMs.

To the best of our knowledge, we are the first to study the imbalanced training of low- and high-frequency identifiers in programming languages. We are also the first to incorporate adversarial training in pre-trained code representation models.

7 Conclusion

In this paper, we study the problem of imbalanced training of low- and high-frequency variable names within pre-trained code representation models. We propose a novel adversarial training strategy, VarGAN, to strengthen the training of rare identifiers. VarGAN treats the code representation model as a generator and introduces an additional classifier to distinguish between low- and high-frequency identifiers. The generator aims to generate vectors that are difficult for the discriminator to judge identifier frequency, thus increasing the overlap between the vector spaces of low- and high-frequency identifiers. We verify the effectiveness of our method in three aspects. First, the identifier vector distribution visualization shows that our method indeed makes the distribution more uniform. Second, the evaluation on IdBench shows that VarGAN achieves the highest correctness in scoring similarity and relatedness for identifier pairs. Finally, evaluation of downstream tasks shows that VarGAN improves code representation models on various tasks, including code summarization and code clone detection.

Our source code and experimental data are publicly available at https://github.com/linyalan2000/VarGAN/.

Acknowledgment

This research is supported by the National Natural Science Foundation of China (Grant No. 62102244) and the National Key Research and Development Program of China (Grant No. 2023YFB4503802).

References

[1] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 38–49, 2015.

[2] S. Chakraborty, T. Ahmed, Y. Ding, P. T. Devanbu, and B. Ray. Natgen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 18–30, 2022.

[3] Q. Chen, J. Lacomis, E. J. Schwartz, G. Neubig, B. Vasilescu, and C. Le Goues. VarCLR: Variable semantic representation pre-training via contrastive learning. In *International Conference on Software Engineering*, ICSE '22, 2022.

[4] N. Chirkova. On the embeddings of variables in recurrent neural networks for source code. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL 2021)*, pages 2679–2689, 2021.

[5] N. Chirkova and S. Troshin. A simple approach for handling out-of-vocabulary identifiers in deep learning for source code. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL 2021)*, pages 278–288, 2021.

[6] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, 2020.

[7] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, pages 321–330, 2008.

[8] C. Gong, D. He, X. Tan, T. Qin, L. Wang, and T.-Y. Liu. Frage: Frequency-agnostic word representation. *Advances in neural information processing systems*, 31, 2018.

[9] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[10] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, L. Shujie, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al. Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations*, 2020.

[11] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[12] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin. Deep code comment generation. In *The 26th conference on program comprehension (ICPC 2018)*, pages 200–210, 2018.

[13] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[14] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes. Big code!= big vocabulary: Open-vocabulary models for source code. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1073–1085, 2020.

[15] J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu. DIRE: A neural approach to decompiled identifier naming. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628–639. IEEE, 2019.

[16] B. Li, H. Zhou, J. He, M. Wang, Y. Yang, and L. Li. On the sentence embeddings from pre-trained language models. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9119–9130, 2020.

[17] F. Liu, G. Li, Z. Fu, S. Lu, Y. Hao, and Z. Jin. Learning to recommend method names with global context. In *The 44th International Conference on Software Engineering (ICSE 2022)*, pages 1294–1306, 2022.

[18] H. Liu, Y. Wang, Z. Wei, Y. Xu, J. Wang, H. Li, and R. Ji. RefBERT: A two-stage pre-trained framework for automatic rename refactoring. *arXiv preprint arXiv:2305.17708*, 2023.

[19] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[20] Z. Liu, X. Xia, D. Lo, Z. Xing, A. E. Hassan, and S. Li. Which variables should I log? *IEEE Transactions on Software Engineering*, 47(9):2012–2031, 2019.

[21] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, et al. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation.

[22] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[23] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI 2016)*, pages 1287–1293, 2016.

[24] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[25] M. Pizka et al. Concise and consistent naming. In *Proceedings. 13th International Workshop on Program Comprehension*, pages 97–106. IEEE Computer Society, 2005.

[26] D. Rattan, R. Bhatia, and M. Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013.

[27] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th international conference on Software engineering*, pages 390–401, 2014.

[28] B. Roziere, M.-A. Lachaux, M. Szafraniec, and G. Lample. Dobf: A deobfuscation pre-training objective for programming languages. *arXiv preprint arXiv:2102.07492*, 2021.

[29] T. Schick and H. Schütze. Bertram: Improved word embeddings have big impact on contextualized model performance. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 3996–4007, 2020.

[30] P. Sedgwick. Spearman's rank correlation coefficient. *Bmj*, 349, 2014.

[31] R. Sennrich, B. Haddow, and A. Birch. Neural machine translation of rare words with subword units. In *54th Annual Meeting of the Association for Computational Linguistics*, pages 1715–1725. Association for Computational Linguistics (ACL), 2016.

[32] J. Shi, Z. Yang, J. He, B. Xu, and D. Lo. Can identifier splitting improve open-vocabulary language model of code? In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2022)*, pages 1134–1138. IEEE, 2022.

[33] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th international conference on software engineering*, pages 378–389, 2014.

[34] J. Siegmund, N. Peitek, C. Parnin, S. Apel, J. Hofmeister, C. Kästner, A. Begel, A. Bethmann, and A. Brechmann. Measuring neural efficiency of program comprehension. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 140–150, 2017.

[35] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the 25th IEEE/ACM international conference on Automated software engineering*, pages 43–52, 2010.

[36] J. Su, J. Cao, W. Liu, and Y. Ou. Whitening sentence representations for better semantics and faster retrieval. *arXiv preprint arXiv:2103.15316*, 2021.

[37] A. Trask, P. Michalak, and J. Liu. Sense2vec - a fast and accurate method for word sense disambiguation in neural word embeddings. *arXiv preprint arXiv:1511.06388*, 2015.

[38] L. Van der Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[39] B. Vasilescu, C. Casalnuovo, and P. Devanbu. Recovering clear, natural identifiers from obfuscated JS names. In *The 2017 11th joint meeting on foundations of software engineering*, pages 683–693, 2017.

[40] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[41] Y. Wainakh, M. Rauf, and M. Pradel. Idbench: Evaluating semantic representations of identifier names in source code. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 562–573. IEEE, 2021.

[42] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, 2021.

[43] F. Yan, M. Li, and Z. Zhou. Towards generating summaries for lexically confusing code through code erosion. In *IJCAI*, pages 3721–3727, 2021.

[44] W. Yu, C. Zhu, Y. Fang, D. Yu, S. Wang, Y. Xu, M. Zeng, and M. Jiang. Dict-bert: Enhancing language model pre-training with dictionary. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 1907–1918, 2022.

[45] T. Zhang, W. Ye, B. Yang, L. Zhang, X. Ren, D. Liu, J. Sun, S. Zhang, H. Zhang, and W. Zhao. Frequency-aware contrastive learning for

neural machine translation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 11712–11720, 2022.

[46] X. Zhou, K. Kim, B. Xu, J. Liu, D. Han, and D. Lo. The devil is in the tails: An exploratory study on long-tailed distributions in software engineering datasets. In *IEEE/ACM International Conference on Automated Software Engineering (ASE 2023)*, 2023.

[47] R. Zhu, L. Yuan, X. Li, M. Gao, and W. Cai. A neural network architecture for program understanding inspired by human behaviors. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL 2022)*, pages 5142–5153, 2022.

Yalan Lin received the B.S. degree from the Department of Computer Science, Xiamen University in 2022. She is currently pursuing the Master's degree in the School of Electronic Information and Electronical Engineering, Shanghai Jiao Tong University. Her research interests include software engineering and natural language processing.

Chengcheng Wan received the B.Eng. degree in Computer Science and Technology from Shanghai Jiao Tong University in 2017 and, Ph.D. degree in Computer Science from the University of Chicago in 2022. She is currently an associate professor at East China Normal University. Her research interests including SE/Sys for AI and AI for SE/Sys.

Shuwen Bai is now pursuing a bachelor's degree in the School of Information Science and Engineering, East China University of Science and Technology.

Xiaodong Gu Xiaodong Gu is currently an associate professor in the School of Software, Shanghai Jiao Tong University. He received a Ph.D. degree from the Department of Computer Science and Engineering from The Hong Kong University of Science and Technology, in 2017. His research interests lie in the broad areas of software engineering and deep learning, including large language models (LLMs) for code, code generation, code search, and code translation.